National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

Non-Stop Monitoring and Debugging on Shared-Memory Multiprocessors

Darlene A. Stewart and W. Morven Gentleman
Software Engineering Group

May 1997

Canada

NRC No. 40147

This paper also appears in *Proceedings of the 2nd Internation Workshop on Software Engineering for Parallel and Distributed Systems*, Boston, MA., May 17-18, 1997, IEEE Computer Society, Los Alamitos, CA., pp. 263–269.

ISBN 0-8186-8043-1

Non-Stop Monitoring and Debugging on Shared-Memory Multiprocessors

Darlene A. Stewart and W. Morven Gentleman
National Research Council of Canada
Montreal Road
Ottawa, ON K1A 0R6
stewart@iit.nrc.ca, gentleman@iit.nrc.ca

Abstract

Monitoring and debugging parallel programs is a difficult activity. There are many situations where the traditional "stop the world, I want to get off" approach to debugging is simply unsuitable. Frequently, nonintrusive monitoring of the program execution is more productive in locating sources of error and also in monitoring "correct" programs for such purposes as performance measurement and tuning.

This paper presents a number of space- and time-efficient tools and techniques to support nonintrusive, non-stop monitoring and debugging of parallel programs running on a shared-memory multiprocessor. The techniques include the use of spy tasks, circular history buffers, vectors of use bits, and data structure audits. Particular emphasis is placed on issues that pertain to parallel computing, such as dealing with concurrent execution, shared memory and data caches.

1. Introduction

Debugging can be defined as the act of locating, analyzing and correcting errors in a program. Debugging usually involves transforming an incorrect program into a correct program, and so it sometimes is referred to as "correctness debugging." On the other hand, *monitoring* is the process of collecting information about a program's execution [1]. One reason to monitor programs is to assist in correctness debugging, but this is not the only reason. Frequently, fully operational, "correct" programs (i.e. those that compute what they are supposed to) are monitored for such purposes as measuring and tuning performance (e.g., processor utilization, disk activity or other resource usage) or suggesting enhancements in future releases. Because the resource usage of a program can be difficult to deduce, using an interactive debugger for "dynamic behaviour monitoring" allows data structures and algorithms to be examined as they work on real data to assist in identifying bottlenecks and in suggesting areas for improvement. Because of the close relationship between debugging and monitoring, it is reasonable to examine tools for both together. The tools and techniques discussed in this paper are applicable not only in correctness debugging but also in monitoring correct programs.

NRC number 40147

It is well recognized that debugging is difficult and that the challenges of debugging parallel programs are even greater than for sequential programs due to the increased complexity introduced by concurrent execution [1]. The majority of debugging and monitoring tools for parallel programs fall into two classes: traditional *cyclical* or *breakpoint* debugging, and *event-based* debugging. Cyclical debugging refers to repeatedly stopping execution of a program to examine the program state and then either continuing the execution or restarting it in order to stop at some earlier point [1]. We also refer to this as the "stop the world, I want to get off" approach to debugging [2]. Event-based debuggers view a parallel program execution as a sequence of events that are collected into a large event history that can later (*post-mortem*) be browsed, analyzed or even replayed to force deterministic execution of a non-deterministic program [1,3,4,5,6]. Much research has been devoted to the analysis and visualization of the excessively large amount of data collected in the event histories [7,8,9,10,11,12,13].

Either of these approaches can be applied to debugging parallel programs with varying degrees of success. However, they suffer from a number of problems which limit their usefulness, particularly for real-time embedded systems. A central problem in debugging parallel programs is the "probe effect" or "Heisenberg uncertainty principle" [1], where just attempting to gather additional information, through breakpointing or other means, may sequentialize or otherwise alter the concurrency of execution just enough to mask insidious bugs, such as critical races. It is truly challenging and frustrating to explore errant behaviour that disappears as soon as one attempts to apply debugging tools. The probe effect seriously hampers the utility of breakpoint debugging due to the intrusive nature of this approach. Indeed, in real-time systems, there may be some performance criteria that the program must meet but is simply unable to in the presence of debugging probes. The “stop the world, I want to get off” approach of breakpoint debuggers is particularly unsuitable in many situations such as: where the program must meet some performance criteria; when there are safety issues involved (e.g., it is unsafe to breakpoint a control program for a robot operating a welding torch); when performing *in situ* stress measurement on a program; or debugging in the field where the program must continue running to provide client services (e.g., telephone switching software).

While event-based debuggers address the probe effect by allowing deterministic replay of non-determinisitic programs, these debuggers still are intrusive, and thus, the probe effect still can be a problem if the event monitoring needs to be turned off for production runs. Also, event-based monitoring/debugging systems with their huge event history traces suffer from problems such as unbounded storage requirements and significant I/O costs.

Segall and Rudolph [14] list support for “programming for observability” as a desired feature in a parallel programming environment. In this paper we discuss ways to provide support for “programming for observability.” The paper explores some tools and techniques for non-stop monitoring and debugging of parallel programs on shared-memory multiprocessors. These tools are particularly effective in an environment where one or more processors can be dedicated to monitoring the activities occurring on the other processors; although use of a dedicated processor is not a requirement, intrusiveness is reduced. We examine the use of spy tasks to glean information about the executing program. We present implementations of circular history buffers and use bits vectors for event recording. We also discuss data structure audit techniques. Throughout the paper, we examine issues about dealing with concurrency, shared memory and processor data caches.

The techniques described in this paper have been implemented in the interactive debugger and dynamic behaviour monitor for Harmony, a multi-tasking, multiprocessor operating system for real-time control [15]. Harmony runs on embedded, shared-memory multiprocessors consisting of a set of single-board computers (Motorola MC68040 or PowerPC processors) on an industry standard backplane bus. Its lightweight threads, which are called *tasks*, communicate and synchronize using a synchronous message passing paradigm (blocking _Send, blocking _Receive, non-blocking _Reply, non-blocking _Try_receive primitives). Scheduling of tasks is preemptive, determined by the priority of the task. A *server* task, in Harmony, is a task that provides some service to other tasks, which are referred to as *clients*. Normally, resources are owned and managed by server tasks instead of being accessed directly.

2. Spy tasks

A very useful and general “programming for observability” technique that can be employed with a great deal of success for nonintrusive debugging on shared-memory multiprocessors is the *spy task*, which we mentioned briefly in [2]. A spy task monitors the execution of one or more other tasks, without stopping them, merely by examining the state of shared global memory. No synchronization is used except the atomicity of memory reads. The interactive debugger user interface itself, if configured as a continuously running task as opposed to only being invoked upon hitting a breakpoint, can be considered to be a spy task. In addition to merely gathering information from shared global memory, a spy task can consolidate the data for presentation, and also it can perform computations on the data to present other quantities of interest to the user. For example, though queue length is not a piece of information stored directly in memory, it may be the role of the spy task to compute and monitor this useful quantity for the user by periodic examination of the queues. Spy tasks are analogous to such things as hardware monitors, line analyzers and snoopy caches.

Use of the simple but powerful spy task technique has been largely overlooked in debugging environments. The Parasight debugger does have the concept of observer programs, called “parasites,” that run in parallel with the target program and monitor its behaviour [16,17]. However, these parasite programs are used primarily to examine and post-process event data that has been explicitly recorded at instrumentation points in the target program, called “scan-points,” rather than to observe the target program data directly. Aral and Gertner [16] also do not expand on the issues of reading data from buffers that are being updated concurrently. Rubin et al. [4] also refer to “spying” on target processors, but in the MAKBILAN architecture, which they describe, there is a monitoring processor shadowing each main processor; beyond the hardware connections, details are not provided on how the monitoring is done.

The use of spy tasks is a powerful technique for minimizing the interference caused by the probe effect. All of the monitoring activity is programmed into a task different from the tasks being monitored or “spied on.”

This spy task then relies on sharing address space with the tasks being monitored to enable it to inobtrusively sample their data while they are running. This technique works particularly well when the processors share address space and the spy task is run on a different processor from the tasks being monitored—the probe effect of the spy task is limited to stealing a few memory cycles. Sampling correlations with the scheduling of the tasks being monitored are also avoided by running the spy task on a different processor.

Modern microprocessors with data caches, such as the Motorola MC68040 or the PowerPC, present a particular challenge for monitoring activities. The usual approach, even in hardware-assisted in-circuit emulators, is to require the data cache to be "turned off" during debugging activities involving memory monitoring. Frequently, this course of action is, of course, unsuitable because of its serious impact on timing; once again, the probe effect is a major factor. Our approach is to debug with the data and instruction caches completely enabled, but to take appropriate precautions so as not to introduce errors by leaving dirty footprints in the data caches.

In a multiprocessor where the data caches are not coherent, the safest way to examine memory is through a *debug agent* running on the target processor being monitored. The debug agent is responsible for reading the requested memory in a safe manner that guarantees that no unexpected footprints are left in the data cache; the agent then returns the read data to the requesting task using the normal message-passing mechanism, which transfers the data in a safe manner. Figure 1 shows the algorithm used by the debug agent to safely read memory; an ownership protocol is assumed whereby data may be left in a processor's cache only if that processor owns the data—ownership must be transferred to another processor before the data can be manipulated by that other processor. By having the debug agent, which runs on the same processor as the tasks being monitored, perform the memory read, we are guaranteed up-to-date values. The price of safety is a slightly greater degree of intrusiveness on the activities being monitored than if the debug agent is not used when spying. However, the probe effect of the debug agent is localized, having only a small impact on CPU and memory cycles around the instant when the data is examined, as compared to running with the caches disabled, which impacts the entire program run in a major way.

```
disable_interrupts;                          /* start of critical section */
flush_cache_lines( address, size );          /*no-op if not already in cache */
memcpy( &buffer, address, size );
invalidate_cache_lines( address, size );
enable_interrupts;                           /* end of critical section */
```

Figure 1. Algorithm used by debug agent to safely read memory with non-coherent caching.

In some situations, it may be worth the risk of the spy task reading "stale" data in order to reduce the interference by minimizing the stolen memory and CPU cycles. As long as the spy task is an observer only (i.e. does not write memory) and is careful to invalidate the data cache line in its own processor after reading by using the same algorithm as the debug agent uses (shown in Figure 1), direct reading of memory by the spy task should cause no adverse caching side effects on the target processor. With some collusion on the part of the observed task, for example, if it pushes "key" values through the cache to memory on updates, the spy task can be assured of current data when it reads the memory directly; the performance hit taken on the observed processor can be much less than in the case of using a debug agent. The need for the debug agent is eliminated in a shared-memory multiprocessor with coherent caching.

Even dynamic data structures can be monitored with only the slightest collusion on the part of the monitored tasks. A technique often employed in spy tasks is the use of optimistic concurrency control to validate the consistency of values read from the data structures that the spy task is monitoring. The monitored task maintains a *generation sequence tag* or *phase number* for the data structure, which it updates each time the data structure is modified. The spy task can read the generation sequence tag, then read the remainder of the data structure, and then read the generation sequence tag again to see if it has changed in order to know whether it has an internally consistent snapshot of the data. Care must obviously be taken in following pointers lest they change underfoot.

3. Circular history buffers

Another non-stop monitoring feature of general utility is the use of circular history buffers for fast intermediate data collection. The use of circular history buffers is not a new concept. They are invented seemingly independently by almost every group of programmers with extensive experience debugging real-time programs. What we have done is to generalize the concept and to address the concurrency issues in parallel programs. Aral and Gertner [16] present a simple example of circular history buffers, but they fail to address the problems associated with the concurrent reading and writing of history buffers. Circular history buffers are a means to overcome the huge volume of data in event traces; usually only the events leading up to some "significant event" are of interest.

To avoid having to rerun the program to learn how some situation occurred, with the consequent questions of determinism or feasibility, we simply record significant "interesting events" as they happen, so that we can go back and examine and post-process them later, if necessary. The events are recorded in a circular buffer avoiding the unbounded storage requirements and the significant costs of external I/O normally associated with event-based monitoring systems. Often data can be recorded in a circular history buffer at times when it is not even possible to perform I/O, say, due to time constraints or the nature of the code from which the data is being recorded. For example, we use the circular history buffer mechanism to monitor the I/O protocol implemented in our multi-window terminal server; in this case it is the I/O system itself that is being monitored. Once a history buffer is completely filled, it wraps around to the start overwriting the oldest events in the buffer, which often are no longer of interest; the frequency of wrapping depends on the frequency of recording, the size of the buffer, and the sizes of the event records. Sometimes multiple buffers are kept with different wrap rates to support the recording of data with differing time scales.

Our implementation offers fast collection of variable length data in multiple circular buffers via the *history buffer server* which supports multiple, named history buffers of user-specified sizes. Data collection from multiple tasks is coordinated; data can be collected into the same buffer even from tasks executing on different processors. All event entries are sequence stamped. The sequence stamp is unique across all the buffers maintained by a given history buffer server so that the sequencing of events across multiple processors and multiple buffers can be resolved; distinct history buffer servers maintain distinct sequence-stamp counters, so history buffers from different servers are not coordinated. Support for multiple, coordinated history buffers of varying sizes facilitates data collection separated according to different wrapping time scales, so that a history buffer collecting one source of data from the program may wrap much more frequently than one collecting another source of data. This allows, for example, coarse-level event collection in one buffer with detailed events collected in a separate buffer that wraps more frequently, thus providing an overall view of the execution plus a fine-detailed view near some significant point of interest.

Separate logging and retrieval tasks (as shown in Figure 2) permit retrieval operations to occur concurrently with recording operations without hindering the speed of recording. This can be an issue if some of the logged messages are exceptionally large, such as screen dumps. An incremental approach is used for retrieving data from a history buffer because the "end of valid data" is a moving target as the oldest data are overwritten by new data being recorded concurrently with the retrieval operation. A retrieval request may be for either the "newest record" or

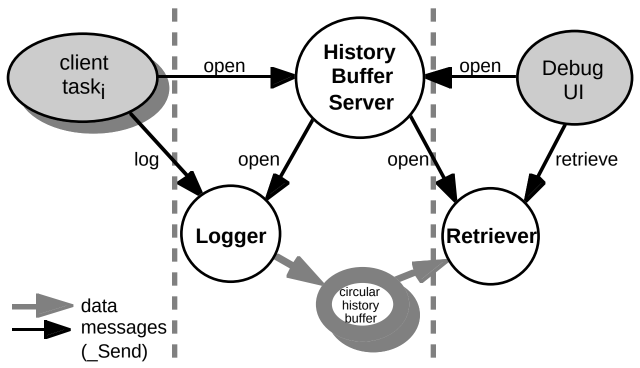

Figure 2. Structure of the history buffer server.

the "record prior to the previous one retrieved". Optimistic concurrency control, using verification on the sequence-stamp field, is employed by the retrieval task to determine whether a record was (partially) overwritten during the interrogation process. The sequence stamp is read and validated before using any pointer or message size information from an event record and again after reading the event record to verify that it has not been invalidated. This facilitates simultaneous logging and retrieval by separate tasks without large amounts of disabled code. The sequence stamp is also used to determine when the oldest record left in the buffer has been retrieved. The history buffer retrieval task is an example of a "spy task" in operation and the concurrent reading and writing of complex data structures.

Care must be taken in logging to support both the variable-sized event records and concurrent retrieval. Each event record contains the following: a sequence stamp, a back pointer to the previous (older) record, the size of the record, the ID of the logging task, a user-defined event type and the event data itself. When a new event is logged, it usually overwrites one or more of the oldest events in the buffer. The event logger is careful to invalidate the sequence stamp of each such event *before* overwriting it (by updating the *oldest-valid-sequence-stamp* field associated with the history buffer); it also covers any unused portion of an overwritten event with a *null-event* record, which has a null sequence stamp.

Tasks may bypass the history buffer server and instead directly access the same library functions used by the history buffer logging/retrieval tasks to record/examine data in circular history buffers. In this case, each recording task must, however, write to a different history buffer. Also, all history buffers sharing a sequence-stamp counter and all tasks recording to those history buffers must reside on the same processor; retrieval tasks need not reside on the same processor as recording tasks. This direct access is considerably faster than using the history buffer server, but it has the obvious limitation of not being able to coordinate events across processor boundaries. Circular history buffers accessed by the less intrusive function interface and examined *post-mortem* have proven to be invaluable in O/S kernel debugging.

In most event-based debuggers, the user has little control over what events are recorded and therefore must rely on post-processing to identify the interesting events in the huge event history traces [4,8,10]; graphical visualization techniques have proven helpful for replaying the events leading up to something [7,11,12,13]. On the other hand, with our circular history buffer scheme, the question of what events to record in the history buffers is up to the application programmer/tester. Event logging is available both compiled in and planted, just as for breakpoints. Experience indicates that recording too much information is even worse than recording too little, as the resultant log may not be long enough, and even if it is, filtering out uninteresting events can be difficult. Determining which “interesting events” are truly interesting is often the challenge.

The fast data collection, user-controlled history buffer size, which may be large or small depending on needs and resources, and the fact that the information remains hidden unless explicitly examined means that it is quite feasible to leave the collection of debugging data into circular buffers permanently in many programs. The probe effect is effectively eliminated under these circumstances. Also, permanently-installed circular history buffers are ideally suited for such activities as *in situ* stress measurement and debugging in the field.

4. Use bits

Another non-stop monitoring facility is the per-processor use-bits vector, mentioned in [2]. This tool is similar to the history buffer but trades temporal resolution for reduced synchronization overhead. The use-bits vector, derived from the use bits and dirty bits of demand-paging systems, provides an inexpensive record of access.

Use bits can be used to identify the occurrence of important events or states in a program, the value of each use bit indicating whether or not the corresponding event or state has occurred. For example, a use bit may indicate whether some piece of code, such as a specific function, has executed at least once, or whether a particular task was ever dispatched. Use bits can also control other activities, such as whether or not the collection of debugging information is to occur.

We chose a macro expansion interface to provide very fast access to set, clear or toggle individual use bits in the vector. The use-bits vector is explicitly allocated; it resides on the same processor as the code writing to it. User control is provided over the size of the use-bits vector and whether it is even allocated. The interface is implemented in such a way that nothing happens if space has not been allocated for the specified use bit. The use-bits vector can be monitored by a spy task, which need not reside on the same processor. The spy task can also clear the entire use-bits vector.

Because of the very fast access and the fact that the use-bits vector need not even be present, it is reasonable to leave the calls to manipulate the use bits in production code. Use bits are an effective, nonintrusive debugging aid, especially for tasks with strict time constraints.

5. Data structure audits

The final “programming for observability” technique we will look at is the use of audit routines for key data structures in the parallel program. Data structure audits have been a standard practise in telephone switches since the 1960’s. However, their use is often overlooked by programmers.

Auditing some data structure for internal consistency is an ideal job for a spy task. As discussed earlier, optimistic concurrency control using generation sequence stamps can be used to verify whether the audit routine has obtained a consistent snapshot of the data structure. A debug agent residing on the same processor as the data structures being interrogated can also assist in reading the data structure in a *safe* manner, particularly for O/S data structures on modern microprocessors with non-coherent data caches. The function of the audit task may be as simple as displaying the data structure for the user, or it may be more sophisticated, performing various processing on the data structure to determine if it is self-consistent and to consolidate large amounts of data for display purposes. Both system and user data structures can be audited in this way.

It is an important function of the multiprocessor debugger to be aware of and to be able to examine various known system data structures, to display relevant information effectively to the user that may help in understanding and debugging his program, and to perform simple consistency checks to aid in early identification of sources of errors. Depending on what we are looking for, these data structures could include: the ready queues on a specified processor, the table of tasks on a specified processor that are awaiting interrupts, the list of all tasks on a processor and their states, the storage pool on a processor, the list of storage blocks allocated to a specified task, etc.

A simple example of automated data structure auditing that we routinely perform is to automatically apply a heuristic to determine if stack overflow may have occurred for some task. This heuristic check is performed each time the user issues the debugger command to list all tasks on a processor along with their states. Each task in the list is checked for stack overflow. The heuristic check itself is very simple—when the stack for a task is created, a "magic value" is written into the end location of the stack; if this magic value is ever overwritten, then in all likelihood, a stack overflow has occurred sometime during execution, and the stack overflow is so flagged in the task display. While there are other explanations for the magic value being overwritten, stack overflow is the most probable cause and certainly the first candidate hypothesis that should be explored. This automated checking can detect stack overflow errors much earlier in program execution and more easily, before too much damage has been done that could lead to very puzzling symptoms.

A more complex example of automated data structure auditing is our safe storage-pool auditor, which checks for corruption of the storage pool on a specified processor. Storage pool corruption can occur as a consequence of such problems as stack overflow, running off the ends of data structures such as arrays, allocating records that are too small, or using uninitialized pointers. Our storage pool is implemented as first fit; the design is preemptable, and storage can be allocated by a preempting task; disable time is bounded. Each memory block in the storage pool is preceded by a header containing: the block size, an indicator of whether the block is allocated or free, and if the block is allocated, an indicator of which task owns it and a link to the other blocks allocated to the same task; if the block is free, part of the block that normally would be available to the user holds a special bit pattern that is unlikely to occur in an allocated block. The storage pool auditor performs consistency checks on these fields for each block in the storage pool and reports any anomalies detected, as well as reporting other interesting information to the user, such as how many blocks of various sizes have been allocated, how much free space is available and a fragmentation map. The storage pool auditor is capable of auditing any storage pool, including the one for the processor on which it runs, while the system continues to run allocating and freeing blocks in the storage pool. The algorithm used is worthy of further attention.

It is not so important that the storage pool auditor display a snapshot of the storage pool at a given instant in time, but rather it is important to walk the pool in a sensible way giving a consistent view of things. To accomplish this the auditor must protect the area of the storage pool in which it is working. We must guarantee that while the storage pool auditor is examining a block, the extent of that block is not altered, i.e. it is not coalesced with another block and no part of it is allocated. We do this by hiding the block so that the storage allocator does not see it when walking the storage pool. This is accomplished by growing another *protected-block*, owned by the storage pool auditor, to cover the block in question; that block is then safely hidden from the storage allocator. Of course, a task owning the block can still free it without any problems; the free routine marks the block as free, and then when the protected-block is eventually shrunk to reveal it again, the storage allocator will see it as a free block.

Walking the storage pool by the auditor proceeds as follows, as illustrated in Figure 3. Starting with a small protected-block reserved at the start of the storage pool for its use, the auditor repeatedly grows the protected-block covering successive blocks in the pool, interrogating, verifying and reporting about each block, until a free block has been covered. If the free block is large enough, it is split and a small portion is taken from the end to be used as the new protected-block, leaving the remainder of the free block available for allocation; otherwise, it is used as-is without being split. Then the previous protected-block is shrunk back to its original size, making all the once-covered blocks visible to the storage allocator. Again, the new protected-block is grown until the next free block is covered. This technique of growing and shrinking protected-blocks is repeated until the end of the storage pool is encountered or a storage pool corruption is detected. The actual updating of the protected-block to cover another block is done through a debug agent running on the processor of the storage pool being audited in order to prevent the storage allocator from manipulating the block being covered during a critical section of the update.

When the storage pool auditor detects a corrupted block, it reports the location of the previous block in the storage pool as well as the corrupted one. This is because the most frequent causes of corruption are due to incorrect

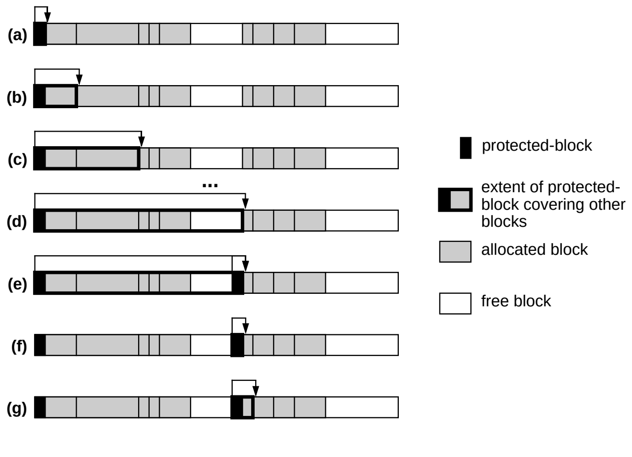

Figure 3. A storage pool being audited.

manipulation of the previous block. Another automated feature of our debugger is to automatically search the memory block lists to find the task owning a particular block, thus facilitating easy identification of the probable culprit task in the case of a storage pool corruption.

The techniques presented above and other similar techniques for auditing data structures are of great utility in debugging parallel programs. Spy tasks, with a little collusion on the part of the tasks owning the data structures being audited and with a little assistance from a debug agent on each target processor, can provide nonintrusive auditing of data structures on shared-memory multiprocessors, even with non-coherent data caches enabled.

6. Summary

This paper has presented a number of mechanisms to support nonintrusive monitoring and debugging of parallel programs running on a shared-memory multiprocessor. The techniques discussed include spy tasks, circular history buffers, vectors of use bits, and automated auditing of system and user data structures. These techniques support the important concept of “programming for observability”. They are general and flexible, and they are both space-efficient and time-efficient. They work particularly well when one or more processors can be dedicated to monitoring the activities of the other processors; although use of a dedicated processor is not a requirement, intrusiveness is reduced. Special emphasis was placed on the issues of dealing with concurrency, shared memory, and processor data caches. Used together, the above techniques are useful in a wide variety of monitoring and debugging situations.

References

[1] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys* 21(4):593-622, December 1989.

[2] W. M. Gentleman and D. A. Stewart. Debugging multi-task programs. In *Proc. Army Research Workshop on Parallel Processing and Medium Scale Multiprocessors*, Stanford, California, January 1986. Also available as NRC/ERB-1008, National Research Council of Canada, Ottawa, Canada.

[3] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers*, C-36(4):471-482, April 1987.

[4] R. Rubin, L. Rudolph, and D. Zernik. Debugging parallel programs in parallel. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):216-225, January 1989.

[5] C.-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):23-34, January 1989.

[6] Y. M. Yong and D. J. Taylor. Performing replay in an OSF DCE environment. In *Proc. CASCON’95*, pages 52-62, Toronto, Ontario, November 1995.

[7] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):163-173, January 1989.

[8] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Ariadne debugger: scalable application of event-based abstraction. In *Proc. ACM SIGPLAN/ONR Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 28(12):85-95, December 1993.

[9] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):11-22, January 1989.

[10] M. Timmerman, F. Gielen, and P. Lambrix. High level tools for the debugging of real-time multiprocessor systems. *Proc. ACM SIGPLAN/ONR Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 28(12):151-157, December 1993.

[11] C. M. Pancake. Customizable portrayals of program structure. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 28(12):64-74, December 1993.

[12] D. Zernik and L. Rudolph. Animating work and time for debugging parallel programs — foundation and experience. *Proc. ACM SIGPLAN/ONR Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 26(12):46-56, December 1991.

[13] D. J. Taylor. The use of process clustering in distributed-system event displays. In *Proc. CASCON’93*, Vol.1:505-512, Toronto, Ontario, October 1993.

[14] Z. Segall and L. Rudolph. Pie: a programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22-37, November 1985.

[15] W. M. Gentleman, S. A. MacKay, D. A. Stewart, and M. Wein. Using the Harmony operating system: Release 3.0, National Research Council of Canada, NRC/ERA-377, Ottawa, Ontario, February 1989.

[16] Z. Aral and I. Gertner. High-level debugging in Parasight. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, *ACM SIGPLAN Notices* 24(1):151-162, January 1989.

[17] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. *Proc. ACM SIGPLAN Symposium on Parallel Programming*, *ACM SIGPLAN Notices* 23(9):21-30, September 1988.